Structuring a Command-Based Robot Project

While users are free to use the command-based libraries however they like (and advanced users are encouraged to do so), new users may want some guidance on how to structure a basic command-based robot project.

A standard template for a command-based robot project is included in the WPILib examples repository (Java, C++). This section will walk users through the structure of this template.

The root package/directory generally will contain four classes:

`Main`, which is the main robot application (Java only). New users *should not* touch this class. `Robot`, which is responsible for the main control flow of the robot code. `RobotContainer`, which holds robot subsystems and commands, and is where most of the declarative robot setup (e.g. button bindings) is performed. `Constants`, which holds globally-accessible constants to be used throughout the robot.

The root directory will also contain two sub-packages/sub-directories: `Subsystems` contains all user-defined subsystem classes. `Commands` contains all user-defined command classes.

Robot

As `Robot` (Java, C++ (Header), C++ (Source)) is responsible for the program's control flow, and command-based is an declarative paradigm designed to minimize the amount of attention the user has to pay to explicit program control flow, the `Robot` class of a command-based project should be mostly empty. However, there are a few important things that must be included

Java

```
  /**
   * This function is run when the
robot is first started up and should
be used for any
   * initialization code.
   */
  @Override
  public void robotInit() {
    // Instantiate our RobotContainer.
This will perform all our button
bindings, and put our
    // autonomous chooser on the
dashboard.
    m_robotContainer = new
RobotContainer();
  }
```

In Java, an instance of `RobotContainer` should be constructed during the `robotInit()` method - this is important, as most of the declarative robot setup will be called from the `RobotContainer` constructor.

In C++, this is not needed as RobotContainer is a value member and will be constructed during the construction of `Robot`.

Java C++ (Source)

```
  /**
   * This function is called every robot packet, no matter the mode. Use this for items like
   * diagnostics that you want ran during disabled, autonomous, teleoperated and test.
   *
   * <p>This runs after the mode specific periodic functions, but before
   * LiveWindow and SmartDashboard integrated updating.
   */
  @Override
  public void robotPeriodic() {
    // Runs the Scheduler.  This is responsible for polling buttons, adding newly-scheduled
    // commands, running already-scheduled commands, removing finished or interrupted commands,
    // and running subsystem periodic() methods.  This must be called from the robot's periodic
    // block in order for anything in the Command-based framework to work.
    CommandScheduler.getInstance().run();
  }
```

The inclusion of the `CommandScheduler.getInstance().run()` call in the `robotPeriodic()` method is essential; without this call, the scheduler will not execute any scheduled commands. Since `TimedRobot` runs with a default main loop frequency of 50Hz, this is the frequency with which periodic command and subsystem methods will be called. It is not recommended for new users to call this method from anywhere else in their code.

Java | C++ (Source)

```
63    /**
64     * This autonomous runs the autonomous
65   command selected by your {@link
66   RobotContainer} class.
67     */
68    @Override
69    public void autonomousInit() {
70      m_autonomousCommand =
71   m_robotContainer.getAutonomousCommand();
72
73      // schedule the autonomous command
74   (example)
      if (m_autonomousCommand != null) {
        m_autonomousCommand.schedule();
      }
    }
```

The `autonomousInit()` method schedules an autonomous command returned by the `RobotContainer` instance. The logic for selecting which autonomous command to run can be handled inside of `RobotContainer`.

Java | C++ (Source)

```
83    @Override
84    public void teleopInit() {
85      // This makes sure that the
86   autonomous stops running when
87      // teleop starts running. If you
88   want the autonomous to
89      // continue until interrupted by
90   another command, remove
91      // this line or comment it out.
92      if (m_autonomousCommand != null) {
        m_autonomousCommand.cancel();
      }
    }
```

The `teleopInit()` method cancels any still-running autonomous commands. This is generally good practice.

Advanced users are free to add additional code to the various init and periodic methods as they see fit; however, it should be noted that including large amounts of imperative robot code in `Robot.java` is contrary to the declarative design philosophy of the command-based paradigm, and can result in confusingly-structured/disorganized code.

RobotContainer

This class (Java, C++ (Header), C++ (Source)) is where most of the setup for your command-based robot will take place. In this class, you will define your robot's subsystems and commands, bind those commands to triggering events (such as buttons), and specify which command you will run in your autonomous routine. There are a few aspects of this class new users may want explanations for:

Java | C++ (Header)

```
24      private final ExampleSubsystem
    m_exampleSubsystem = new
    ExampleSubsystem();
```

Notice that subsystems are declared as private fields in `RobotContainer`. This is in stark contrast to the previous incarnation of the command-based framework, but is much more-aligned with agreed-upon object-oriented best-practices. If subsystems are declared as global variables, it allows the user to access them from anywhere in the code. While this can make certain things easier (for example, there would be no need to pass subsystems to commands in order for those commands to access them), it makes the control flow of the program much harder to keep track of as it is not

immediately obvious which parts of the code can change or be changed by which other parts of the code. This also circumvents the ability of the resource-management system to do its job, as ease-of-access makes it easy for users to accidentally make conflicting calls to subsystem methods outside of the resource-managed commands.

Java | C++ (Source)

```
26    private final ExampleCommand
      m_autoCommand = new
      ExampleCommand(m_exampleSubsystem);
```

Since subsystems are declared as private members, they must be explicitly passed to commands (a pattern called “dependency injection”) in order for those commands to call methods on them. This is done here with `ExampleCommand`, which is passed a pointer to an `ExampleSubsystem`.

Java | C++ (Source)

```
38    /**
39     * Use this method to define your butto
40    mappings.  Buttons can be created by
41     * instantiating a {@link GenericHID} o
42    subclasses ({@link
43     * edu.wpi.first.wpilibj.Joystick} or t
44    XboxController}), and then passing it to
45     * {@link
      edu.wpi.first.wpilibj2.command.button.Joy
       */
      private void configureButtonBindings()
      }
```

As mentioned before, the `RobotContainer()` constructor is where most of the declarative setup for the robot should take place, including button bindings, configuring autonomous selectors, etc. If the constructor gets too "busy," users are encouraged to migrate code into separate subroutines (such as the `configureButtonBindings()` method included by default) which are called from the constructor.

Java | C++ (Source)

```
48    /**
49     * Use this to pass the autonomous
50    command to the main {@link Robot}
51    class.
52     *
53     * @return the command to run in
54    autonomous
55     */
56    public Command
57    getAutonomousCommand() {
        // An ExampleCommand will run in
      autonomous
        return m_autoCommand;
      }
    }
```

Finally, the `getAutonomousCommand()` method provides a convenient way for users to send their selected autonomous command to the main `Robot` class (which needs access to it to schedule it when autonomous starts).

Constants

The `Constants` class (Java, C++ (Header)) (in C++ this is not a class, but simply a header file in which several namespaces are defined) is where globally-accessible robot constants (such as speeds, unit

conversion factors, PID gains, and sensor/motor ports) can be stored. It is recommended that users separate these constants into individual inner classes corresponding to subsystems or robot modes, to keep variable names shorter.

In Java, all constants should be declared `public static final` so that they are globally accessible and cannot be changed. In C++, all constants should be `constexpr`.

For more illustrative examples of what a `constants` class should look like in practice, see those of the various command-based example projects:

- FrisbeeBot (Java, C++)
- GyroDriveCommands (Java, C++)
- Hatchbot (Java, C++)

In Java, it is recommended that the constants be used from other classes by statically importing the necessary inner class. An `import static` statement imports the static namespace of a class into the class in which you are working, so that any `static` constants can be referenced directly as if they had been defined in that class. In C++, the same effect can be attained with `using namespace`:

Java C++

```
import static
edu.wpi.first.wpilibj.templates.commandbased.Const
```

Subsystems

User-defined subsystems should go in this package/directory.

Commands

User-defined commands should go in this package/directory.